## Chapter 1. What is Spring Web Services?

## 1.1. Introduction

Spring Web Services (Spring-WS) is a product of the Spring community focused on creating document-driven Web services. Spring Web Services aims to facilitate contract-first SOAP service development, allowing for the creation of flexible web services using one of the many ways to manipulate XML payloads. The product is based on Spring itself, which means you can use the Spring concepts such as dependency injection as an integral part of your Web service.

People use Spring-WS for many reasons, but most are drawn to it after finding alternative SOAP stacks lacking when it comes to following Web service best practices. Spring-WS makes the best practice an easy practice. This includes practices such as the WS-I basic profile, Contract-First development, and having a loose coupling between contract and implementation. The other key features of Spring Web services are:

**Powerful mappings.** You can distribute incoming XML requests to any object, depending on message payload, SOAP Action header, or an XPath expression.

**XML API support.** Incoming XML messages can be handled not only with standard JAXP APIs such as DOM, SAX, and StAX, but also JDOM, dom4j, XOM, or even marshalling technologies.

**Flexible XML Marshalling.** The Object/XML Mapping module in the Spring Web Services distribution supports JAXB 1 and 2, Castor, XMLBeans, JiBX, and XStream. And because it is a separate module, you can use it in non-Web services code as well.

**Reuses your Spring expertise.** Spring-WS uses Spring application contexts for all configuration, which should help Spring developers get up-to-speed nice and quickly. Also, the architecture of Spring-WS resembles that of Spring-MVC.

**Supports WS-Security.** WS-Security allows you to sign SOAP messages, encrypt and decrypt them, or authenticate against them.

**Integrates with Acegi Security.** The WS-Security implementation of Spring Web Services provides integration with [Acegi Security](http://acegisecurity.org). This means you can use your existing Acegi configuration for your SOAP service as well.

**Built by Maven.** This assists you in effectively reusing the Spring Web Services artifacts in your own Maven-based projects.

**Apache license.** You can confidently use Spring-WS in your project.

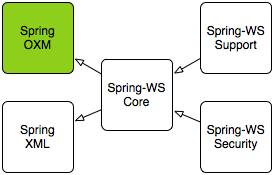
## 1.2. Runtime environment

Spring Web Services requires a standard Java 1.5 Runtime Environment. Java 1.6 is also supported. Spring-WS also requires Spring 3.0 or higher.

Spring-WS consists of a number of modules, which are described in the remainder of this section.

* The XML module (spring-xml.jar) contains various XML support classes for Spring Web Services. This module is mainly intended for the Spring-WS framework itself, and not a Web service developers.
* The Core module (spring-ws-core.jar) is the central part of the Spring's Web services functionality. It provides the central [WebServiceMessage](http://docs.spring.io/spring-ws/site/reference/html/common.html#web-service-messages) and [SoapMessage](http://docs.spring.io/spring-ws/site/reference/html/common.html#soap-message) interfaces, the [server-side](http://docs.spring.io/spring-ws/site/reference/html/server.html) framework, with powerful message dispatching, and the various support classes for implementing Web service endpoints; and the [client-side](http://docs.spring.io/spring-ws/site/reference/html/client.html) WebServiceTemplate.
* The Support module (spring-ws-support.jar) contains additional transports (JMS, Email, and others).
* The [Security](http://docs.spring.io/spring-ws/site/reference/html/security.html) package (spring-ws-security.jar) provides a WS-Security implementation that integrates with the core Web service package. It allows you to add principal tokens, sign, and decrypt and encrypt SOAP messages. Additionally, it allows you to leverage your existing Spring Security security implementation for authentication and authorization.

The following figure illustrates the Spring-WS modules and the dependencies between them. Arrows indicate dependencies, i.e. Spring-WS Core depends on Spring-XML and the OXM module found in Spring 3.



Dependencies between Spring-WS modules

**There are two approaches**

1. **Contract First :** developing web services that start with the XML Schema/WSDL contract first followed by the Java code second.
2. **Contact Last**

You have the idea what these approaches are all about. Now the important thing is to choose one out these.

Spring only support the Contract First approach because of some reasons here <http://docs.spring.io/spring-ws/site/reference/html/why-contract-first.html>

**2.3. Contract-first versus Contract-last**

Besides the Object/XML Mapping issues mentioned in the previous section, there are other reasons for preferring a contract-first development style.

**2.3.1. Fragility**

As mentioned earlier, the contract-last development style results in your web service contract (WSDL and your XSD) being generated from your Java contract (usually an interface). If you are using this approach, you will have no guarantee that the contract stays constant over time. Each time you change your Java contract and redeploy it, there might be subsequent changes to the web service contract.

Aditionally, not all SOAP stacks generate the same web service contract from a Java contract. This means changing your current SOAP stack for a different one (for whatever reason), might also change your web service contract.

When a web service contract changes, users of the contract will have to be instructed to obtain the new contract and potentially change their code to accommodate for any changes in the contract.

In order for a contract to be useful, it must remain constant for as long as possible. If a contract changes, you will have to contact all of the users of your service, and instruct them to get the new version of the contract.

### 2.3.2. Performance

When Java is automatically transformed into XML, there is no way to be sure as to what is sent across the wire. An object might reference another object, which refers to another, etc. In the end, half of the objects on the heap in your virtual machine might be converted into XML, which will result in slow response times.

When using contract-first, you explicitly describe what XML is sent where, thus making sure that it is exactly what you want.

**2.3.3. Reusability**

Defining your schema in a separate file allows you to reuse that file in different scenarios. If you define an AirportCode in a file called airline.xsd, like so:

<simpleType name="AirportCode">

<restriction base="string">

<pattern value="[A-Z][A-Z][A-Z]"/>

</restriction>

</simpleType>

You can reuse this definition in other schemas, or even WSDL files, using an import statement.

### 2.3.4. Versioning

Even though a contract must remain constant for as long as possible, they do need to be changed sometimes. In Java, this typically results in a new Java interface, such as AirlineService2, and a (new) implementation of that interface. Of course, the old service must be kept around, because there might be clients who have not migrated yet.

If using contract-first, we can have a looser coupling between contract and implementation. Such a looser coupling allows us to implement both versions of the contract in one class. We could, for instance, use an XSLT stylesheet to convert any "old-style" messages to the "new-style" messages.

**The goodness of spring official tutorials is the reason because of which the chapter 3 is copied as it is and highlighted for good approaches.**

## Chapter 3. Writing Contract-First Web Services

## 3.1. Introduction

This tutorial shows you how to write [contract-first Web services](http://docs.spring.io/spring-ws/site/reference/html/why-contract-first.html), that is, developing web services that start with the XML Schema/WSDL contract first followed by the Java code second. Spring-WS focuses on this development style, and this tutorial will help you get started. Note that the first part of this tutorial contains almost no Spring-WS specific information: it is mostly about XML, XSD, and WSDL. The [second part](http://docs.spring.io/spring-ws/site/reference/html/tutorial.html#tutorial-creating-project) focuses on implementing this contract using Spring-WS .

The most important thing when doing contract-first Web service development is to try and think in terms of XML. This means that Java-language concepts are of lesser importance. It is the XML that is sent across the wire, and you should focus on that. The fact that Java is used to implement the Web service is an implementation detail. An important detail, but a detail nonetheless.

In this tutorial, we will define a Web service that is created by a Human Resources department. Clients can send holiday request forms to this service to book a holiday.

## 3.2. Messages

In this section, we will focus on the actual XML messages that are sent to and from the Web service. We will start out by determining what these messages look like.

### 3.2.1. Holiday

In the scenario, we have to deal with holiday requests, so it makes sense to determine what a holiday looks like in XML:

<Holiday xmlns="http://mycompany.com/hr/schemas">

<StartDate>2006-07-03</StartDate>

<EndDate>2006-07-07</EndDate>

</Holiday>

A holiday consists of a start date and an end date. We have also decided to use the standard [ISO 8601](http://www.cl.cam.ac.uk/%7Emgk25/iso-time.html) date format for the dates, because that will save a lot of parsing hassle. We have also added a namespace to the element, to make sure our elements can used within other XML documents.

### 3.2.2. Employee

There is also the notion of an employee in the scenario. Here is what it looks like in XML:

<Employee xmlns="http://mycompany.com/hr/schemas">

<Number>42</Number>

<FirstName>Arjen</FirstName>

<LastName>Poutsma</LastName>

</Employee>

We have used the same namespace as before. If this <Employee/> element could be used in other scenarios, it might make sense to use a different namespace, such as http://mycompany.com/employees/schemas.

### 3.2.3. HolidayRequest

Both the holiday and employee element can be put in a <HolidayRequest/>:

<HolidayRequest xmlns="http://mycompany.com/hr/schemas">

<Holiday>

<StartDate>2006-07-03</StartDate>

<EndDate>2006-07-07</EndDate>

</Holiday>

<Employee>

<Number>42</Number>

<FirstName>Arjen</FirstName>

<LastName>Poutsma</LastName>

</Employee>

</HolidayRequest>

The order of the two elements does not matter: <Employee/> could have been the first element just as well. What is important is that all of the data is there. In fact, the data is the only thing that is important: we are taking a data-driven approach.

## 3.3. Data Contract

Now that we have seen some examples of the XML data that we will use, it makes sense to formalize this into a schema. This data contract defines the message format we accept. There are four different ways of defining such a contract for XML:

* DTDs
* [XML Schema (XSD)](http://www.w3.org/XML/Schema)
* [RELAX NG](http://www.relaxng.org/)
* [Schematron](http://www.schematron.com/)

DTDs have limited namespace support, so they are not suitable for Web services. Relax NG and Schematron certainly are easier than XML Schema. Unfortunately, they are not so widely supported across platforms. We will use XML Schema.

By far the easiest way to create an XSD is to infer it from sample documents. Any good XML editor or Java IDE offers this functionality. Basically, these tools use some sample XML documents, and generate a schema from it that validates them all. The end result certainly needs to be polished up, but it's a great starting point.

Using the sample described above, we end up with the following generated schema:

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"

elementFormDefault="qualified"

targetNamespace="http://mycompany.com/hr/schemas"

xmlns:hr="http://mycompany.com/hr/schemas">

<xs:element name="HolidayRequest">

<xs:complexType>

<xs:sequence>

<xs:element ref="hr:Holiday"/>

<xs:element ref="hr:Employee"/>

</xs:sequence>

</xs:complexType>

</xs:element>

<xs:element name="Holiday">

<xs:complexType>

<xs:sequence>

<xs:element ref="hr:StartDate"/>

<xs:element ref="hr:EndDate"/>

</xs:sequence>

</xs:complexType>

</xs:element>

<xs:element name="StartDate" type="xs:NMTOKEN"/>

<xs:element name="EndDate" type="xs:NMTOKEN"/>

<xs:element name="Employee">

<xs:complexType>

<xs:sequence>

<xs:element ref="hr:Number"/>

<xs:element ref="hr:FirstName"/>

<xs:element ref="hr:LastName"/>

</xs:sequence>

</xs:complexType>

</xs:element>

<xs:element name="Number" type="xs:integer"/>

<xs:element name="FirstName" type="xs:NCName"/>

<xs:element name="LastName" type="xs:NCName"/>

</xs:schema>

This generated schema obviously can be improved. The first thing to notice is that every type has a root-level element declaration. This means that the Web service should be able to accept all of these elements as data. This is not desirable: we only want to accept a <HolidayRequest/>. By removing the wrapping element tags (thus keeping the types), and inlining the results, we can accomplish this.

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"

xmlns:hr="http://mycompany.com/hr/schemas"

elementFormDefault="qualified"

targetNamespace="http://mycompany.com/hr/schemas">

<xs:element name="HolidayRequest">

<xs:complexType>

<xs:sequence>

<xs:element name="Holiday" type="hr:HolidayType"/>

<xs:element name="Employee" type="hr:EmployeeType"/>

</xs:sequence>

</xs:complexType>

</xs:element>

<xs:complexType name="HolidayType">

<xs:sequence>

<xs:element name="StartDate" type="xs:NMTOKEN"/>

<xs:element name="EndDate" type="xs:NMTOKEN"/>

</xs:sequence>

</xs:complexType>

<xs:complexType name="EmployeeType">

<xs:sequence>

<xs:element name="Number" type="xs:integer"/>

<xs:element name="FirstName" type="xs:NCName"/>

<xs:element name="LastName" type="xs:NCName"/>

</xs:sequence>

</xs:complexType>

</xs:schema>

The schema still has one problem: with a schema like this, you can expect the following messages to validate:

<HolidayRequest xmlns="http://mycompany.com/hr/schemas">

<Holiday>

<StartDate>this is not a date</StartDate>

<EndDate>neither is this</EndDate>

</Holiday>

*<!-- ... -->*

</HolidayRequest>

Clearly, we must make sure that the start and end date are really dates. XML Schema has an excellent built-in date type which we can use. We also change the NCNames to strings. Finally, we change the sequence in <HolidayRequest/> to all. This tells the XML parser that the order of <Holiday/> and <Employee/> is not significant. Our final XSD now looks like this:

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"

xmlns:hr="http://mycompany.com/hr/schemas"

elementFormDefault="qualified"

targetNamespace="http://mycompany.com/hr/schemas">

<xs:element name="HolidayRequest">

<xs:complexType>

<xs:all>

<xs:element name="Holiday" type="hr:HolidayType"/> (1)

<xs:element name="Employee" type="hr:EmployeeType"/>

</xs:all>

</xs:complexType>

</xs:element>

<xs:complexType name="HolidayType">

<xs:sequence>

<xs:element name="StartDate" type="xs:date"/>

<xs:element name="EndDate" type="xs:date"/> (2)

</xs:sequence> (2)

</xs:complexType>

<xs:complexType name="EmployeeType">

<xs:sequence>

<xs:element name="Number" type="xs:integer"/>

<xs:element name="FirstName" type="xs:string"/>

<xs:element name="LastName" type="xs:string"/> (3)

</xs:sequence> (3)

</xs:complexType>

</xs:schema>

|  |  |  |
| --- | --- | --- |
| 1 |  | all tells the XML parser that the order of <Holiday/> and <Employee/> is not significant. |
| 2 |  | We use the xsd:date data type, which consist of a year, month, and day, for <StartDate/> and <EndDate/>. |
| 3 |  | xsd:string is used for the first and last name. |

We store this file as hr.xsd.

## 3.4. Service contract

A service contract is generally expressed as a [WSDL](http://www.w3.org/TR/wsdl) file. Note that in Spring-WS, writing the WSDL by hand is not required. Based on the XSD and some conventions, Spring-WS can create the WSDL for you, as explained in the section entitled [Section 3.6, “Implementing the Endpoint”](http://docs.spring.io/spring-ws/site/reference/html/tutorial.html#tutorial.implementing.endpoint). You can skip to [the next section](http://docs.spring.io/spring-ws/site/reference/html/tutorial.html#tutorial-creating-project) if you want to; the remainder of this section will show you how to write your own WSDL by hand.

We start our WSDL with the standard preamble, and by importing our existing XSD. To separate the schema from the definition, we will use a separate namespace for the WSDL definitions: http://mycompany.com/hr/definitions.

<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"

xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

xmlns:schema="http://mycompany.com/hr/schemas"

xmlns:tns="http://mycompany.com/hr/definitions"

targetNamespace="http://mycompany.com/hr/definitions">

<wsdl:types>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:import namespace="http://mycompany.com/hr/schemas" schemaLocation="hr.xsd"/>

</xsd:schema>

</wsdl:types>

Next, we add our messages based on the written schema types. We only have one message: one with the <HolidayRequest/> we put in the schema:

<wsdl:message name="HolidayRequest">

<wsdl:part element="schema:HolidayRequest" name="HolidayRequest"/>

</wsdl:message>

We add the message to a port type as an operation:

<wsdl:portType name="HumanResource">

<wsdl:operation name="Holiday">

<wsdl:input message="tns:HolidayRequest" name="HolidayRequest"/>

</wsdl:operation>

</wsdl:portType>

That finished the abstract part of the WSDL (the interface, as it were), and leaves the concrete part. The concrete part consists of a binding, which tells the client how to invoke the operations you've just defined; and a service, which tells it where to invoke it.

Adding a concrete part is pretty standard: just refer to the abstract part you defined previously, make sure you use document/literal for the soap:binding elements (rpc/encoded is deprecated), pick a soapAction for the operation (in this case http://mycompany.com/RequestHoliday, but any URI will do), and determine the location URL where you want request to come in (in this case http://mycompany.com/humanresources):

<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"

xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"

xmlns:schema="http://mycompany.com/hr/schemas"

xmlns:tns="http://mycompany.com/hr/definitions"

targetNamespace="http://mycompany.com/hr/definitions">

<wsdl:types>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:import namespace="http://mycompany.com/hr/schemas" (1)

schemaLocation="hr.xsd"/>

</xsd:schema>

</wsdl:types>

<wsdl:message name="HolidayRequest"> (2)

<wsdl:part element="schema:HolidayRequest" name="HolidayRequest"/> (3)

</wsdl:message>

<wsdl:portType name="HumanResource"> (4)

<wsdl:operation name="Holiday">

<wsdl:input message="tns:HolidayRequest" name="HolidayRequest"/> (2)

</wsdl:operation>

</wsdl:portType>

<wsdl:binding name="HumanResourceBinding" type="tns:HumanResource"> (4)(5)

<soap:binding style="document" (6)

transport="http://schemas.xmlsoap.org/soap/http"/> (7)

<wsdl:operation name="Holiday">

<soap:operation soapAction="http://mycompany.com/RequestHoliday"/> (8)

<wsdl:input name="HolidayRequest">

<soap:body use="literal"/> (6)

</wsdl:input>

</wsdl:operation>

</wsdl:binding>

<wsdl:service name="HumanResourceService">

<wsdl:port binding="tns:HumanResourceBinding" name="HumanResourcePort"> (5)

<soap:address location="http://localhost:8080/holidayService/"/> (9)

</wsdl:port>

</wsdl:service>

</wsdl:definitions>

|  |  |
| --- | --- |
| 1 | We import the schema defined in [Section 3.3, “Data Contract”](http://docs.spring.io/spring-ws/site/reference/html/tutorial.html#tutorial.xsd). |
| 2 | We define the HolidayRequest message, which gets used in the portType. |
| 3 | The HolidayRequest type is defined in the schema. |
| 4 | We define the HumanResource port type, which gets used in the binding. |
| 5 | We define the HumanResourceBinding binding, which gets used in the port. |
| 6 | We use a document/literal style. |
| 7 | The literal http://schemas.xmlsoap.org/soap/http signifies a HTTP transport. |
| 8 | The soapAction attribute signifies the SOAPAction HTTP header that will be sent with every request. |
| 9 | The http://localhost:8080/holidayService/ address is the URL where the Web service can be invoked. |

This is the final WSDL. We will describe how to implement the resulting schema and WSDL in the next section.

## 3.5. Creating the project

In this section, we will be using [Maven3](http://maven.apache.org/) to create the initial project structure for us. Doing so is not required, but greatly reduces the amount of code we have to write to setup our HolidayService.

The following command creates a Maven3 web application project for us, using the Spring-WS archetype (that is, project template)

mvn archetype:create -DarchetypeGroupId=org.springframework.ws \

-DarchetypeArtifactId=spring-ws-archetype \

-DarchetypeVersion=2.1.4.RELEASE \

-DgroupId=com.mycompany.hr \

-DartifactId=holidayService

This command will create a new directory called holidayService. In this directory, there is a 'src/main/webapp' directory, which will contain the root of the WAR file. You will find the standard web application deployment descriptor 'WEB-INF/web.xml' here, which defines a Spring-WS MessageDispatcherServlet and maps all incoming requests to this servlet.

<web-app xmlns="http://java.sun.com/xml/ns/j2ee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee

http://java.sun.com/xml/ns/j2ee/web-app\_2\_4.xsd"

version="2.4">

<display-name>MyCompany HR Holiday Service</display-name>

*<!-- take especial notice of the name of this servlet -->*

<servlet>

<servlet-name>***spring-ws***</servlet-name>

<servlet-class>org.springframework.ws.transport.http.MessageDispatcherServlet</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>spring-ws</servlet-name>

<url-pattern>/\*</url-pattern>

</servlet-mapping>

</web-app>

In addition to the above 'WEB-INF/web.xml' file, you will also need another, Spring-WS-specific configuration file, named 'WEB-INF/spring-ws-servlet.xml'. This file contains all of the Spring-WS-specific beans such as EndPoints, WebServiceMessageReceivers, and suchlike, and is used to create a new Spring container. The name of this file is derived from the name of the attendant servlet (in this case 'spring-ws') with '-servlet.xml' appended to it. So if you defined a MessageDispatcherServlet with the name 'dynamite', the name of the Spring-WS-specific configuration file would be 'WEB-INF/dynamite-servlet.xml'.

(You can see the contents of the 'WEB-INF/spring-ws-servlet.xml' file for this example in [???](http://docs.spring.io/spring-ws/site/reference/html/tutorial.html" \l "tutorial.example.sws-conf-file).)

Once you had the project structure created, you can put the schema and wsdl from previous section into 'WEB-INF/' folder.

## 3.6. Implementing the Endpoint

In Spring-WS, you will implement Endpoints to handle incoming XML messages. An endpoint is typically created by annotating a class with the @Endpoint annotation. In this endpoint class, you will create one or more methods that handle incoming request. The method signatures can be quite flexible: you can include just about any sort of parameter type related to the incoming XML message, as will be explained later.

### 3.6.1. Handling the XML Message

In this sample application, we are going to use [JDom](http://www.jdom.org) to handle the XML message. We are also using [XPath](http://www.w3schools.com/xpath/), because it allows us to select particular parts of the XML JDOM tree, without requiring strict schema conformance.

package com.mycompany.hr.ws;

import java.text.SimpleDateFormat;

import java.util.Date;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.ws.server.endpoint.annotation.Endpoint;

import org.springframework.ws.server.endpoint.annotation.PayloadRoot;

import org.springframework.ws.server.endpoint.annotation.RequestPayload;

import com.mycompany.hr.service.HumanResourceService;

import org.jdom.Element;

import org.jdom.JDOMException;

import org.jdom.Namespace;

import org.jdom.xpath.XPath;

@Endpoint (1)

public class HolidayEndpoint {

private static final String NAMESPACE\_URI = "http://mycompany.com/hr/schemas";

private XPath startDateExpression;

private XPath endDateExpression;

private XPath nameExpression;

private HumanResourceService humanResourceService;

@Autowired

public HolidayEndpoint(HumanResourceService humanResourceService) (2)

throws JDOMException {

this.humanResourceService = humanResourceService;

Namespace namespace = Namespace.getNamespace("hr", NAMESPACE\_URI);

startDateExpression = XPath.newInstance("//hr:StartDate");

startDateExpression.addNamespace(namespace);

endDateExpression = XPath.newInstance("//hr:EndDate");

endDateExpression.addNamespace(namespace);

nameExpression = XPath.newInstance("concat(//hr:FirstName,' ',//hr:LastName)");

nameExpression.addNamespace(namespace);

}

@PayloadRoot(namespace = NAMESPACE\_URI, localPart = "HolidayRequest") (3)

public void handleHolidayRequest(@RequestPayload Element holidayRequest) (4)

throws Exception {

SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");

Date startDate = dateFormat.parse(startDateExpression.valueOf(holidayRequest));

Date endDate = dateFormat.parse(endDateExpression.valueOf(holidayRequest));

String name = nameExpression.valueOf(holidayRequest);

humanResourceService.bookHoliday(startDate, endDate, name);

}

}

|  |  |
| --- | --- |
| 1 | The HolidayEndpoint is annotated with @Endpoint. This marks the class as a special sort of @Component, suitable for handling XML messages in Spring-WS, and also making it eligible for suitable for component scanning. |
| 2 | The HolidayEndpoint requires the HumanResourceService business service to operate, so we inject the dependency via the constructor and annotate it with @Autowired. Next, we set up XPath expressions using the JDOM API. There are three expressions: //hr:StartDate for extracting the <StartDate> text value, //hr:EndDate for extracting the end date and concat(//hr:FirstName,' ',//hr:LastName) for extracting and concatenating the names of the employee. |
| 3 | The @PayloadRoot annotation tells Spring-WS that the handleHolidayRequest method is suitable for handling XML messages. The sort of message that this method can handle is indicated by the annotation values, in this case, it can handle XML elements that have the HolidayRequest local part and the http://mycompany.com/hr/schemas namespace. More information about mapping messages to endpoints is provided in the next section. |
| 4 | The handleHolidayRequest(..) method is the main handling method method, which gets passed with the <HolidayRequest/> element from the incoming XML message. The @RequestPayload annotation indicates that the *holidayRequest* parameter should be mapped to the payload of the request message. We use the XPath expressions to extract the string values from the XML messages, and convert these values to Date objects using a SimpleDateFormat. With these values, we invoke a method on the business service. Typically, this will result in a database transaction being started, and some records being altered in the database. Finally, we define a void return type, which indicates to Spring-WS that we do not want to send a response message. If we wanted a response message, we could have returned a JDOM Element that represents the payload of the response message. |

Using JDOM is just one of the options to handle the XML: other options include DOM, dom4j, XOM, SAX, and StAX, but also marshalling techniques like JAXB, Castor, XMLBeans, JiBX, and XStream, as is explained in the next chapter. We chose JDOM because it gives us access to the raw XML, and because it is based on classes (not interfaces and factory methods as with W3C DOM and dom4j), which makes the code less verbose. We use XPath because it is less fragile than marshalling technologies: we don't care for strict schema conformance, as long as we can find the dates and the name.

Because we use JDOM, we must add some dependencies to the Maven pom.xml, which is in the root of our project directory. Here is the relevant section of the POM:

<dependencies>

<dependency>

<groupId>org.springframework.ws</groupId>

<artifactId>spring-ws-core</artifactId>

<version>2.1.4.RELEASE</version>

</dependency>

<dependency>

<groupId>jdom</groupId>

<artifactId>jdom</artifactId>

<version>1.0</version>

</dependency>

<dependency>

<groupId>jaxen</groupId>

<artifactId>jaxen</artifactId>

<version>1.1</version>

</dependency>

</dependencies>

Here is how we would configure these classes in our spring-ws-servlet.xml Spring XML configuration file, by using component scanning. We also instruct Spring-WS to use annotation-driven endpoints, with the <sws:annotation-driven> element.

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:sws="http://www.springframework.org/schema/web-services"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/web-services http://www.springframework.org/schema/web-services/web-services-2.0.xsd

http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<context:component-scan base-package="com.mycompany.hr"/>

<sws:annotation-driven/>

</beans>

### 3.6.2. Routing the Message to the Endpoint

As part of writing the endpoint, we also used the @PayloadRoot annotation to indicate which sort of messages can be handled by the handleHolidayRequest method. In Spring-WS, this process is the responsibility of an EndpointMapping. Here we route messages based on their content, by using a PayloadRootAnnotationMethodEndpointMapping. The annotation used above:

@PayloadRoot(namespace = "http://mycompany.com/hr/schemas", localPart = "HolidayRequest")

basically means that whenever an XML message is received with the namespace http://mycompany.com/hr/schemas and the HolidayRequest local name, it will be routed to the handleHolidayRequest method. By using the <sws:annotation-driven> element in our configuration, we enable the detection of the @PayloadRoot annotations. It is possible (and quite common) to have multiple, related handling methods in an endpoint, each of them handling different XML messages.

There are also other ways to map endpoints to XML messages, which will be described in the next chapter.

### 3.6.3. Providing the Service and Stub implementation

Now that we have the Endpoint, we need HumanResourceService and its implementation for use by HolidayEndpoint.

package com.mycompany.hr.service;

import java.util.Date;

public interface HumanResourceService {

void bookHoliday(Date startDate, Date endDate, String name);

}

For tutorial purposes, we will use a simple stub implementation of the HumanResourceService.

package com.mycompany.hr.service;

import java.util.Date;

import org.springframework.stereotype.Service;

@Service (1)

public class StubHumanResourceService implements HumanResourceService {

public void bookHoliday(Date startDate, Date endDate, String name) {

System.out.println("Booking holiday for [" + startDate + "-" + endDate + "] for [" + name + "] ");

}

}

|  |  |
| --- | --- |
| 1 | The StubHumanResourceService is annotated with @Service. This marks the class as a business facade, which makes this a candidate for injection by @Autowired in HolidayEndpoint. |

## 3.7. Publishing the WSDL

Finally, we need to publish the WSDL. As stated in [Section 3.4, “Service contract”](http://docs.spring.io/spring-ws/site/reference/html/tutorial.html#tutorial-service-contract), we don't need to write a WSDL ourselves; Spring-WS can generate one for us based on some conventions. Here is how we define the generation:

<sws:dynamic-wsdl id="holiday" (1)

portTypeName="HumanResource" (3)

locationUri="/holidayService/" (4)

targetNamespace="http://mycompany.com/hr/definitions"> (5)

<sws:xsd location="/WEB-INF/hr.xsd"/> (2)

</sws:dynamic-wsdl>

|  |  |
| --- | --- |
| 1 | The id determines the URL where the WSDL can be retrieved. In this case, the id is holiday, which means that the WSDL can be retrieved as holiday.wsdl in the servlet context. The full URL will typically be http://localhost:8080/holidayService/holiday.wsdl. |
| 3 | Next, we set the WSDL port type to be HumanResource. |
| 4 | We set the location where the service can be reached: /holidayService/. We use a relative URI and we instruct the framework to transform it dynamically to an absolute URI. Hence, if the service is deployed to different contexts we don't have to change the URI manually. For more information, please refer to [Section 5.3.1.1, “Automatic WSDL exposure”](http://docs.spring.io/spring-ws/site/reference/html/server.html#server-automatic-wsdl-exposure)  For the location transformation to work, we need to add an init parameter to spring-ws servlet in web.xml:  <init-param>  <param-name>transformWsdlLocations</param-name>  <param-value>true</param-value>  </init-param> |
| 5 | We define the target namespace for the WSDL definition itself. Setting this attribute is not required. If not set, the WSDL will have the same namespace as the XSD schema. |
| 2 | The xsd element refers to the human resource schema we defined in [Section 3.3, “Data Contract”](http://docs.spring.io/spring-ws/site/reference/html/tutorial.html#tutorial.xsd). We simply placed the schema in the WEB-INF directory of the application. |

You can create a WAR file using **mvn install**. If you deploy the application (to Tomcat, Jetty, etc.), and point your browser at [this location](http://localhost:8080/holidayService/holiday.wsdl), you will see the generated WSDL. This WSDL is ready to be used by clients, such as [soapUI](http://www.soapui.org/), or other SOAP frameworks.

That concludes this tutorial. The tutorial code can be found in the full distribution of Spring-WS. The next step would be to look at the echo sample application that is part of the distribution. After that, look at the airline sample, which is a bit more complicated, because it uses JAXB, WS-Security, Hibernate, and a transactional service layer. Finally, you can read the rest of the reference documentation.